STOS Compiler

EmuMovies

USER GUIDE

STOS Compiler

François Lionet	*STOS Compiler programmer*
Richard Vanner	*Project manager*
Chris Payne	*Marketing manager*
Stephen Hill	*Manual author*
David McLachlan	*Graphics*

If you have any difficulty with this product, please write to:
Mandarin Software,
Europa House, Adlington Park,
Adlington, Macclesfield SK10 4NP

No material may be reproduced in whole or in part without written permission. While every care has been taken, the publishers cannot be held legally responsible for any errors or omissions in the manual or the software.

Contents

Foreword by François Lionet

I would like to take this opportunity to thank you for your continued support of the STOS Basic package. I do hope you like this compiler, as I put all my programming knowledge and four months of my life into it. Rest assured that Jawx, Mandarin and I will do the maximum possible to help you with any questions or difficulties. We'll also keep on developing the STOS range to increase its power still further. So do have fun writing games, compiling them, and hopefully even selling them. Finally, please go on supporting us, and don't give this compiler to other people. MERCI BEAUCOUP, as we say in France. Try to imagine how you would feel, if someone were to steal your latest STOS masterpiece, especially if programming was your only source of income. And remember, software piracy isn't just a game, it is a genuine threat to the entire software industry.
I've only one other thing to say: Try compiling the compiler! Happy compiling.

François

Introduction

The original STOS Basic package set a new standard for Atari ST software. Now, **hot** on the heels of its phenomenal success, comes this amazing utility which can transform any existing STOS Basic program into incredibly fast machine code. The new STOS compiler gives you all the speed of a language like C, with the ease of use you have come to expect from STOS Basic. Unlike the compilers for other Basics, the STOS compiler is completely interactive. So you can compile, run and test your programs directly from STOS Basic. You can also create standalone programs which can be executed straight from the Gem desktop. These programs can be freely distributed without any copyright restrictions.

It's important to note that the STOS Compiler has been designed especially with 520ST users in mind. This means you can compile full-sized STOS Basic programs on an unexpanded machine with absolutely no disc swapping! Even on the smallest system, you will easily be able to compile all the programs from the STOS Games disc.

This package is delightfully easy to use: All the features are controlled directly from the mouse, using a simple accessory program. Compared to the complexities of programs like the Sprite editor, or the Character generator, the compiler is remarkably uncomplicated. But don't be deceived – the STOS compiler is a very sophisticated program indeed.

Compilers versus Interpreters

But what exactly is a compiler? As you may know, the ST's 68000 processor is only capable of understanding an internal language known as machine code. This means that you cannot simply enter Basic commands straight into the ST without first translating them into machine-code using STOS Basic.

Supposing you were presented with some valuable text written in an unfamiliar language (say French). There are two possible ways this could be translated into English. One idea might be to take each word in the text, and look it up separately in a French-English dictionary.

This approach is known as interpretation, and is used by the standard STOS Basic. Whenever a program is run, each instruction is checked against a dictionary stored somewhere in the ST's memory, and the appropriate machine code routine is subsequently executed.

Another solution to the above problem might be to hand the text over to a professional translator. This person could now rewrite the entire document in English, which you could then read directly. The same idea can also be applied to a Basic program. In this case, the translator corresponds to a separate program known as a compiler. The STOS Basic compiler converts the complete STOS Basic program into machine-code, producing a new version which can be executed immediately without the need for further translation.

The main advantages of a compiled program over an interpreted one can be summarised like this:

1 Compiled programs execute up to three times faster than the equivalent interpreted program.

2 Compiled programs can be run directly from the Gem Desktop, or from an AUTO folder. They do not need any of the files from the STOS folder in order to run.

3 Once a program has been compiled, it is impossible to translate it back into STOS Basic. So there is no chance of anyone stealing your original code.

Against this, there is one single disadvantage:

Compiled programs are larger than interpreted programs.

This statement is actually slightly misleading because the real size of an interpreted program is far larger than you would initially expect. Take the game Orbit, for instance. Although this is apparently only 60k, if you were to create a run only version you would need to include all the separate program modules contained in the STOS folder. The total size of the program would therefore be a surprising 270k.

Contrast this with a compiled version of the same program. The final length of Orbit after compilation is around 130k, which is over 140k less than the interpreted version! So although a compiled program may look larger than an interpreted one, it's often considerably smaller.

1: Installation

Before you can use your STOS Basic compiler for the first time, you will need to configure it for your ST. Although the configuration process may seem a little complicated, it can normally be completed in under ten minutes, and only needs to be done once. It's important to emphasise that the compiler itself is incredibly easy to use. So if you have already mastered the intricacies of the STOS Sprite editor, this package will hold no terrors for you! You should begin by making a backup of the system on a fresh disc. Once you've created this disc, you should use it for all subsequent compilation. You can now hide the original disc somewhere safe, secure in the knowledge that you can make another copy of the compiler if the backup gets corrupted.

Backing up the Compiler disc

1 Slide the write protect tab of the original disc so that you can see through the hole. This will guard your disc against possible mistakes during copying.

2 Place a blank disc into drive A and format it in the normal way.

3 Now put the compiler disc into drive A and drag the icon over drive B.

4 Follow the prompts displayed in the Gem dialogue boxes.

Note that this package was only designed to run under STOS Basic version 2.4 or higher. You don't need to panic if you have an earlier version, as an upgrade to 2.4 is included free with the compiler.

The main improvements incorporated into version 2.4 are:
- Better support for extension files.
- The Floating point arithmetic now uses single precision, and is **much** faster.

- A few minor bugs have been fixed. Ninety per cent of existing STOS Basic programs wil be totally compatible with the new version. But any programs which use floating arithmetic will need to be converted to STOS 2.4 using the CONVERT.BAS program supplied with this disc. See Chapter 6 for further details.

Installing STOS V2.4 onto a floppy disc system

1 Boot up STOS Basic as normal.
2 Place the compiler disc into drive A.
3 Enter the line:

```
run "stosv204.bas"
```

4 Select the current drive using the A and B keys, and hit G to load the new STOS files into the ST's memory. You can now place a disc containing STOS Basic into the appropriate drive. We recommend that you update ALL your copies of STOS to version 2.4 as this will avoid any potential mix ups with the compiler. But don't update your original copy of the STOS language disc until AFTER you have successfully copied one of your backups, and tested it carefully. Otherwise a single corrupted file on the compiler disc could accidentally destroy all your copies of STOS in one fell swoop!
5 Repeat step 4 for all your working copies of STOS Basic.

Installing STOS V2.4 on a hard disc

1 Create a floppy version of STOS V2.4 using the above procedure.

2 Copy the STOS folder onto the hard disc along with the BASIC204.PRG file.

Auto-loading the compiler

The action of the compiler is controlled through the accessory program COMPILER.ACB. This can be loaded automatically by changing the EDITOR.ENV file on the STOS boot disc.
Insert the original STOS language disc into drive A and run the configuration program CONFIG by typing:

```
run "config.bas"
```

When the main menu appears, click on the NEXT PAGE icon. You can now add COMPILER.ACB to the list of accessories which will be loaded on start-up. Click the mouse on the first free space of the accessory list and enter the line:

```
compiler.acb
```

It's a good idea to add the following function key definitions to the standard list. This will simplify the process of loading and saving compiled programs considerably.

```
F14(shift-F4) fload"*.CMP"
F15(shift-F5) fsave"*.CMP"
```

You can now save the new configuration file onto your working copy of STOS Basic using the "SAVE ON DISC" command.Then copy the COMPILER.ACB file onto the language disc so that STOS Basic can pick it up off the root directory as it boots up.

Using the compiler

The compiler accessory can be executed at any time directly from the <HELP> menu. In order for the compiler to work, the files contained in the COMPILER folder should always be available from the current drive. This reduces the amount of memory used by the compiler to a mere 25k, and allows you to compile acceptably large STOS Basic programs on a 520 ST.

The optimum strategy for using this package varies depending on your precise system configuration. Here is a full explanation of how the package can be set up for use with most common ST systems.

Using the compiler on a 512k ST with one floppy drive

If you are intending to compile large programs on an unexpanded machine, it's wise to keep a copy of the COMPILER folder on every disc you will be using for your programs. This will allow you to compile large programs directly onto the disc, without the risk of running out of memory. A special program called COMPCOPY is supplied for this purpose, which automatically copies the appropriate files onto your discs. Insert the compiler disc into drive A and type:

```
accload "compcopy"
```

Press HELP and select COMPCOPY with the appropriate function key. Now press G to load the entire contents of the folder into the ST's memory. You will then be prompted for a blank disc, which should be placed into drive A, and the compiler files will be copied onto your new disc.

Depending on the format of your drive, you will be left with either 200k or 480k on each disc. This should prove more than adequate for all but the largest STOS programs. Despite this, it's still possible that you will occasionally run out of memory. See the troubleshooting section at the end of this chapter if you have any problems.

Incidentally, the STOS compiler does NOT allow you to swap discs during the compilation process. So don't try to compile a program from drive A to drive B if you are limited to a single drive.

Using the compiler on a 512k ST with two floppy drives

When you use the compiler, place a disc containing the COMPILER folder into drive A, and your program disc in drive B. This will provide you with plenty of disc space to compile even the largest STOS programs.

Using the compiler with a ramdisc (1040ST or higher)

You can increase the speed of the STOS Basic compiler significantly by copying the contents of the COMPILER folder onto a ramdisc. We have therefore included a special STOS-compatible ramdisc along with the compiler. This can be created using the STOSRAM.ACB accessory from the disc.

1 Load STOSRAM from the compiler disc with the line:

```
accnew:accload "stosram.acb"
```

Enter the accessory by pressing <HELP> and then<F1>
2 Choose the size of your ramdisc by pressing the S key and entering the number of kilobytes you require. Note that the minimum space required to hold the entire contents of the compiler folder is 150k, and this is why the default setting is 150.
3 You must now set the full path name of the folder which will be loaded from the disc during initialisation. This can be done with the C option. We have set the default to A:\COMPILER but you can set it to any other name you require.
4 Finally, insert a disc containing both STOS Basic, and the COMPILER folder into drive A. Now hit G to add a ramdisc to the existing AUTO folder. This ramdisc will subsequently be created whenever STOS Basic is loaded. On start-up the entire contents of the compiler folder will be automatically copied to the new ramdisc.

The speed increase to be gained from using the compiler in this way is literally staggering. Typical compilation speeds are an amazing 10k per second. This means that the BULLET program from the STOS GAME disc can be compiled in well under 15 seconds! For further details of the STOSRAM accessory see chapter 5.

For users with a single density disc drive it's important to realise that the STOS language disc and the COMPILER folder won't both fit on a 320k disc. To solve this, copy the STOS language disc first and remove one of the picture files from the STOS folder. If you're using a colour monitor then remove the PIC.P13 file otherwise remove the PIC.P11 picture. This will ensure that enough space is available for copying the COMPILER folder.

Using the compiler with a hard disc

The default path name for the compiler folder is normally set from the first line of the compiler accessory. This can be changed to any directory you wish by editing the accessory like so:

```
load "compiler.acb"
10 COMPATH$="D:\STOS\UTILITY":rem Example path name
save"compiler.acb"
```

You can now copy over the COMPILER folder into the required directory of your hard disc. Note that if the COMPTEST string is empty (default), the accessory uses the following strategy to find the COMPILER folder.

1 The current directory is searched.
2 The accessory checks the root directory of the present drive.
3 The root directories of drives any available drives from C upwards are examined.

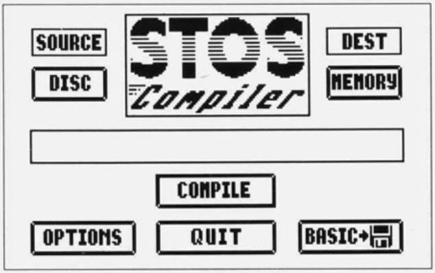
2: The Compiler accessory

If you found the installation procedure rather cumbersome, you will be delighted to hear that the compilation process is simplicity itself. You begin by booting up STOS Basic using the new configuration file. This will automatically load the COMPILER accessory into the ST's memory during initialisation.

Alternatively you can also load the accessory directly from the compiler disc using the line:

accload "compiler"

You can now enter the compiler accessory from the <HELP> menu in the normal way, The following control panel will be displayed on the ST's screen:



The main features of the compiler are controlled through a set of five "buttons". These can be activated by simply moving the mouse pointer over the appropriate area and clicking once on the left mouse key.

SOURCE

The SOURCE button is used to determine whether a program is to be compiled either from memory or the current disc. Clicking on the box underneath toggles between the two possibilities:

MEMORY This option informs the compiler that you wish to compile the program you are currently editing. Any of the four program segments may be compiled independently without affecting the contents of the others. Compiling from memory is **very** fast, but it does consume a large amount of memory.

DISC Some programs are simply too large to be compiled directly from memory. In these circumstances it's convenient to compile a program from a file on the disc. Obviously this is slower than the memory option, but most STOS programs can still easily be compiled within a matter of minutes. Before using this feature, remember to ensure that the COMPILER folder is accessible from the current drive. Also note that the DISC option will be selected automatically whenever memory is running short.

DEST

The DEST button selects the eventual destination of the compiled program. Program's may be compiled either to memory or directly into a file on the disc.

MEMORY This option compiles the program into memory. Note that the memory used by this feature is completely separate from your original program. So you can subsequently save the compiled program onto the disc without erasing your current STOS Basic program.

DISC If you choose the DISC as the destination, the code will be compiled straight into a file without taking up any valuable memory. Since it's much slower than the MEMORY directive, it's only really suitable for compiling particularly large STOS Basic programs.

COMPILE

The compilation process is started when you click on the COMPILE button with the mouse. As your program is compiled, a horizontal bar grows across the screen. When this completes its journey, the compilation has been successfully concluded. But if an error is detected, the compiler will terminate and you will be returned to the STOS Basic editor.

Occasionally, errors will be generated from supposedly bug-free programs like Zoltar. The reason for these errors is that the interpreter is only capable of detecting an error in the line which is currently being run. So if an error exists in a section of code which is rarely executed, it can easily be missed. Since the compiler tests the **whole** program, rather than just the current line, it is able to discover all the syntax errors in your program at once.

QUIT

Exits from the compiler accessory and returns you to the editor.

DISC

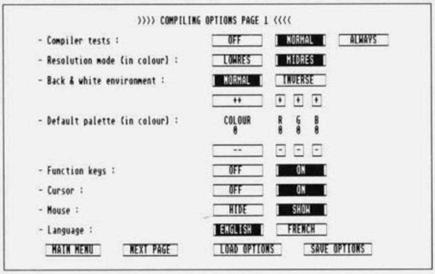
This button allows you to choose whether the compiled program is to be run either from STOS Basic, or directly from the Gem Desktop.

BASIC
This is the default, and generates a compiled program which can only be run within the STOS Basic system. Files produced with this option have the extension ".CMP".

GEM
The GEM directive allows you to create a program which can be run independently of the STOS Basic system. These programs have the extension ".PRG", and can only be executed from the Gem Desktop. Furthermore, since they consist entirely of machine code, they cannot be listed or amended from STOS Basic. Programs in this format can be sold or distributed in any way you like. Depending on the facilities used, the Gem run version of a file will be between 40 and 80k larger than the equivalent STOS run program.

OPTIONS

Whenever the compiler is loaded, a number of configuration settings are read from a special OPTIONS.INF file in the COMPILER folder. These settings provide you with the ability to fine tune the eventual program to your particular needs. They can be changed at any time by simply clicking on the OPTIONS button from the compiler menu which displays the following screen:



```
>>>> COMPILING OPTIONS PAGE 1 <<<<

- Compiler tests :                 OFF    [✓ NORMAL]   ALWAYS
- Resolution mode (in colour) :    LOWRES [✓ MIDRES]
- Back & white environment :       [✓ NORMAL]  INVERSE
                                   ++      +  +  +
- Default palette (in colour) :    COLOUR  R  G  B
                                   0       0  0  0
                                   --      -  -  -
- Function keys :                  OFF    [✓ ON]
- Cursor :                         OFF    [✓ ON]
- Mouse :                          HIDE   [✓ SHOW]
- Language :                       [✓ ENGLISH]  FRENCH

  MAIN MENU    NEXT PAGE    LOAD OPTIONS    SAVE OPTIONS
```

COMPILER TESTS

This option is used to set the frequency of certain internal checks. Although the coordinates of a STOS sprite are updated using interrupts, the sprites on the screen are only moved prior to the execution of a Basic instruction. While this is happening, STOS also checks for CONTROL+ C, and tests whether the pull- down menus have been accessed by the user.

COMPILER TEST OFF: This completely removes the tests from the compiled code. The result is that the compiled program ignores CONTROL+C, refuses to open your menus, and does not automatically update your sprites when they are moved. Set against this, however, is the fact that the final code will be around 10% faster.

COMPILER TEST NORMAL: A check is performed before every branch such asGOTO, NEXT, REPEAT, WEND, ELSE and THEN. Tests are also carried out prior to slow instructions such as PRINT and INPUT. This setting is used as thedefault.

COMPILER TEST ALWAYS: Adds a test before every STOS Basic instruction, leading to particularly smooth sprite movements. As you would expect, programs compiled in this way are slightly slower than with NORMAL or OFF settings.

Note that these settings can also be changed directly from your STOS Basic programs. See Chapter 5 for further details.

Gem-run options

These options allow you to tailor the default environment of a compiled program whichis to be run from the Desktop. They have no effect on any programs compiled for use within STOS Basic.

RESOLUTION MODE: This directive allows you to select between low ormediumresolution when your program is executed from the Desktop using a colour monitor. To change the resolution simply click on the appropriate icon. Note that if your program is subsequently run on a monochrome monitor, this option is completely ignored.

BLACK AND WHITE ENVIRONMENT: Chooses between normal or inverse graphics when a Gem-run program is executed on a monochrome monitor.
NORMAL Uses white text on a black background.
INVERSE Produces a "paper white" display.

DEFAULT PALETTE: This allows you to assign the colours which will be initially used for your graphics.
The first icons select one of the 16 possible colours to be set (4 in medium resolution)

++	Click on this box to increment the colour number by one
COLOUR 0	
- -.	Click here to decrement the colour number by one

The rightmost icon buttons set the exact hue of this colour.

+	+	+
R	G	B
0	0	0
-	-	-

Click on a "+" to add one to the red, green or blue component to the colour respectively.

"-" buttons subtract one from the appropriate colour value.

For speed you can quickly step through the values by pressing in the right mousebutton, but for subtle single steps use the left button. This applies for any other optionthat uses '-' and '+' icons.

FUNCTION KEYS: The window used for the STOS function key assignments wil normally be drawn on the screen during the initialisation process. This adds a slightly unprofessional feel to your finished program. You can avoid this effect using theollowing directive from the compiler menu.

ON	The function key window is automatically drawn at the top of the screen during initialisation.
OFF	The function key window is omitted.

CURSOR: Activates or deactivates the text cursor when the program is initialised. This setting can be changed at any time from within your compiled program using the CURS ON/OFF commands.

MOUSE: The MOUSE option allows you to decide whether the mouse will be turned on or off as a default. As you might expect, you can reactivate the mouse within your program using the STOS Basic SHOW instruction.

LANGUAGE: Toggles the language used for any system messages between ENGLISH and FRENCH.

MAIN MENU: Returns you to the main compiler menu.

NEXT PAGE: Displays the next page of options.(See below)

LOAD OPTIONS: Loads an existing set of options from an OPTIONS.INF file fromthe disc.

SAVE OPTIONS: Saves the current options to an OPTIONS.INF file in the COMPILER directory.

LOADED CHARACTER SETS: The compiled program normally includes all three of the STOS character sets. But if you only intend to run your program in a single resolution, the character sets used by the remaining modes will waste valuable memory. The STOS compiler therefore lets you select precisely which character sets are to be loaded.
Warning! Any attempt to run your program in the wrong resolution after this option has been set, will crash the ST completely.

LOADED MOUSE POINTERS: As with the character sets, you can use this option `to omit the data used by the mouse pointers in the resolutions your program will not be using. But be careful, as improper use of this command can crash the ST!

WINDOW BUFFER SIZE: The windowing system used by STOS Basic, normally keeps a copy of the entire contents of all the windows which your program has defined.
If yourprogram doesn't use windows, then this memory will be completely wasted. The default setting is 32k. This can be altered in 1k steps by clicking on the "++" and "- -" boxes. You can calculate the memory needed by your windows using the following simple rules:

Each character position takes up two bytes in medium/high resolution and four bytes in low resolution. In low resolution the main STOS screen holds 1000 characters, so the memory taken by this screen is 1000*4 or 4000 bytes.

If you change this setting, don't forget about the function key window, and the file selector! These use about 8k of memory.

SPRITE BUFFER SIZE: Before a STOS sprite is copied to the screen, it is first drawn into a separate memory buffer. If you are really pressed for space and you are only using the smallest sprites, you can reduce this buffer to around 1k.
*Warning! This option is extremely dangerous. Do **not** use it unless you know precisely what you are doing, or the ST will almost certainly crash!*

3: Compiler tutorial

I'll now go through the process of using the compiler in a little more detail. You start off by loading the compiler accessory into memory with a line like:

```
accload "compile"
```

Now enter the following STOS program into your computer:

```
10 timer=0:for i=1 to 100000:next i
20 print "Loop took ";timer/50.0;" seconds"
```

This program will take approximately seven seconds to run using interpreted STOS Basic. Insert a disc containing the compiler folder in drive A, and enter the accessory menu with the <HELP> key. The compiler can now accessed by selecting the COMPILER accessory.

Move the pointer over the COMPILE button and click on the mouse. The disc will now whir for a few seconds as the required compiler libraries are accessed from the disc. As the program is translated, the progress of the compiler is represented by a horizontal bar. When this reaches the edge of the screen, the compilation process has been successfully completed. You will now be presented with the option to grab your finished program into one of the available program segments.

Position the mouse pointer over the first free segment and click on the left button. Note that selecting Save from this menu displays a standard STOS file selector which can be used to save your compiled program straight to the disc. Thecompiler will now GRAB the compiled program into the free program area selected.

Compiled programs are executed using the familiar RUN command from STOS Basic,. so just type RUN<RETURN> This performs in around three seconds which is over twice the speed of the interpreted version.

Incidentally, since the program has been converted into machine code, any attempt to generate a listing will produce the response:

```
**********************************
*                                *
*   COMPILED PROGRAM             *
*   Don't change line 65535!     *
*                                *
**********************************
```

Line 65535 contains a special instruction which executes the compiled program stored in the ST's memory. Removing this line will effectively destroy your compiled program. It's theoretically possible to incorporate separate lines of interpreted Basic into the compiled program. This practice is **not** however, recommended.
Note that compiled programs can also be accessed using the normal LOAD and SAVE instructions. If you wanted to save your current program, you could therefore type something like:

```
save "loop.cmp"
```

One unique feature of the STOS compiler, is that you can keep your interpreted program in memory while you are debugging the compiled version. Whenever a bug is detected, you can then effortlessly flick back to the Basic code, and make the appropriate changes. This code can be subsequently re-compiled in a matter of seconds, without having to leave the STOS Basic environment at all.

The previous example was relatively trivial. I'll now show you how a full-sized game can be compiled with this system.

Place the STOS games disc into drive A and type:

```
dir:rem Update current disc directory
dir$="\bullet":rem Enter Bullet directory
load "bullet.bas":rem Load Bullet
```

Now insert a copy of the compiler disc into drive A. If you're using an unexpanded 520 ST, this disc should have been created previously using the COMPCOPY program I mentioned earlier.

First call the COMPILER accessory from the <HELP> menu in the normal way. When the main screen appears, click on the button immediately below DEST. This will force the compiled program to be generated directly onto the disc, and may be omitted if you are using a 1040 ST. Now choose the COMPILE option and click once on the left mouse button. You will then be presented with a standard STOS file selector which prompts you for the name of your compiled program. Enter a name like "BULLET.CMP".

After a few minutes, the compilation process will be completed, and you will be returned to the compiler screen. Exit from the accessory using the QUIT option and type:

```
accnew:rem Remove all accessories (only needed for 520 users)
load "bullet.cmp"
```

Now place a copy of the STOS Games disc and enter the lines:

```
dir:rem Update directory
dir$="\bullet"
run
```

Your newly compiled version of Bullet Train will now execute in the usual way. As you can see, compiled programs are much faster than interpreted ones!
So far, I've only shown you how to create a compiled program for use within STOS Basic. But the ability to generate Gem runable programs is much more exciting as it enables you to distribute your work with none of the protection problems encountered with a run-only interpreted program.

I'll begin with a small example which displays a Neochrome picture on the ST's screen. Type in the following program:

```
5 mode 0:flash off
10 F$=file select$("*.NEO","Display a NEOCHROME screen")
20 if F$="" or len(F$)<5 then end
30 if right$(F$,4)<>".NEO" then boom : goto 10
40 hide:load F$,back:rem Load screen
50 wait key:show
```

Put your working copy of the compiler disc into drive A and call up the compiler from the <HELP> menu. Now click on the box marked BASIC. The title of this box should immediately change to GEM. You can now start the compilation process with the COMPILE button. After a short while, you will be prompted for a filename for your new program. This file will then be written to the disc and the compilation process will be concluded.

If you wish to test this program, you will need to leave STOS completely and execute it from the Gem Desktop. Don't forget to save your original program first!

On average, Gem-run programs are about 40k larger than the equivalent compiled program. The reason for the increase in size is that Gem-run programs have to be completely self sufficient. This requires them to incorporate large segments of the appropriate compiler libraries.

Gem-run programs can be run directly from the Desktop like any other program. They do not need any support from the rest of the STOS system. Note that once you have compiled a program in this format, it cannot be subsequently executed from the STOS Basic system. You should therefore **always** retain a copy of your program in its original interpreted form.

Finally, I'll provide you with a full-sized example of a Gem-run program. Place a disc containing the sprite editor definer into drive A and load it with:

```
dir:rem Update current directory
load "sprite.acb"
```

Enter a disc containing the compiler libraries into drive A. Since this is very large program there won't be enough space to compile it directly into memory on a 520 ST. In this case you should specify compilation from memory to disc by clicking on the button below DEST. Now toggle the BASIC-> icon to GEM->, and select COMPILE. The disc will be accessed for a couple of minutes as the sprite editor is compiled onto the disc.

You have now produced a complete stand-alone version of the sprite editor which can be run independently of the STOS system. This might well prove extremely useful, especially when you are importing graphics directly from the Neochrome or Degas drawing packages.

4: Troubleshooting

Although you are unlikely to encounter any major problems when using the compiler, it's still possible that an unforeseen difficulty could occur at one time or another. We've therefore provided you with a comprehensive troubleshooting guide which will help you through most of the more common errors.

The compiler generates an out of memory error

This can happen if you are trying to compile a large (100k+) program on an unexpanded 520 ST. The compiler provides you with four different options which can be used to conserve memory. Here is a list of the various possibilities in descending order of speed:

SOURCE	DESTINATION	COMMENTS
MEMORY	MEMORY	Very fast but uses the maximum amount ofmemory
DISC	MEMORY	Slower but uses considerably less memory
MEMORY	DISC	Slightly slower than disc to memory but the memory usage can occasionally be less.
DISC	DISC	Uses very little memory.

The only limit to the size of your program is the amount of available disc space. This is quite slow on a single floppy. When you get an out of memory error, you should try each of the above options in turn. If you still have problems, you will need to reduce the size of your program in some way.

The easiest solution is to get rid of the permanent memory banks which are used by the program. These can be defined during program initialisation using the RESERVE command and loaded separately from the disc. Your programs initialisation phase will now include the following steps.

1 Define each screen bank with RESERVE AS SCREEN.

2 Load these screens from the disc with LOAD.

3 Define any DATA banks in the original program as WORK banks. Use the RESERVE AS WORK command for this purpose.

4 Load the external data from the disc.

Since a large percentage of the space used by many STOS programs is taken up by the memory banks, this technique can lead to dramatic reductions in size, without noticeably affecting the programs performance.

Another idea is to split your program into several parts, and load these into memory with RUN when required. This technique is known as overlay programming, and is commonly used in commercial games. If you do use this approach, you will need to remember to compile each program module separately. Don't try to combine interpreted modules with compiled modules or your program will fail when run from the desktop.

The compiler returns an UNDIMENSIONED ARRAY ERROR for an array which has apparently been correctly dimensioned.

The compiler requires the DIM statement to occur in the listing BEFORE the arrays are used. Take the following example.

```
10 gosub 1000
20 a(10)=50
30 end
1000 dim A(100):return
```

This causes an error because when the compiler checks line 20, it has yet to encounter the DIM statement at line 1000. It therefore generates an erroneous error message. The solution to this problem is simply to dimension all arrays at the start of the program. So you can fix the above routine by replacing line 10 with:

```
10 dim a(100)
```

You get a syntax error at an ON...GOTO or ON...GOSUB statement

In order to optimise the speed of the compiler, the line numbers used by ON GOTO and ON GOSUB are required to use constants rather than variables.

So a line like:

```
on A goto 1000,10000+A*5,500
```

will produce a syntax error. This should be replaced by:

```
on A goto 1000,10010,500
```

A previously error-free program returns a syntax error when compiled

This happens quite often, and is simply a reflection of the improved sensitivity of the compiler to genuine syntax errors. Take the following program:

```
10 print "hi there"
20 goto 10
30 prunt "This is an error"
```

If you try to run this program using the interpreter, then the spelling mistake at line 30 will be missed, since it is never actually executed. But if you compile it, the compiler will detect the error immediately and ask you to correct it.

Problems occur when you try to compile a program using certain extension commands

Any extensions which are to be compiled need to have a separate extension file for the compiler. This has the extension ".ECN" where N is the identifier of the extension file. The appropriate file will normally be included along with the extensions, and should be always be placed in the COMPILER folder.

The colours of a Gem-run program are different from the interpreted version

This problem can occur if you have been altering the default colour settings using the options menu. Remember that when these are saved to the disc, they affect all subsequent compilation. Correct by simply restoring the standard options from the original compiler disc.

A program which reserves a memory bank within a FOR...NEXT loop crashes inexplicably

A program which creates a memory bank within a FOR...NEXT loop will behave unpredictably if the bank number is held in an array. This could lead to a total crash of the STOS Basic system. The reasons for these problems are complex, but the sort of code to watch out for is:

```
10 dim b(15)
20 for b(3)=1 to 10
30 reserve as screen b(3)
40 next b(3)
```

The difficulty can be avoided by either using a simple variable as the index, or defining the banks explicitly outside the FOR...NEXT. For example:

```
20 for i=1 to 10
30 reserve as screen i
40 next i
```

5: Compiler extension commands

The compiler adds three extended commands to the normal STOS Basic system. These commands are only used in a compiled program. They have no effect whatsoever when the code is interpreted.

In a normal STOS Basic program the following tests are performed at regular intervals.

- Sprite updates.
- Menu checks.
- Control+ C tests.

The COMPTEST instructions provide you with fine control over the testing process.

COMPTEST ON

Checks are only carried out before jump instructions such as GOTO and WHILE, and especially slow commands like PRINT or WAIT. Note that COMPTEST ON is the default setting used by interpreted programs.

COMPTEST OFF

The COMPTEST OFF command stops the testing completely, improving the speed of the program by up to 10%. This allows you to optimize time critical sections of a compiled program. It is particularly useful for routines which have to perform large numbers of complex calculations in a relatively short space of time. Typical examples of such programs include 3D graphics packages and fractal generators. One dangerous side effect of this command is that it is impossible to interrupt a program until the compiler tests are restored. So try to get into the habit of saving your current program before calling this function. Otherwise, an infinite loop could lock up the system completely, losing all your valuable data.

Example:

```
10 dim a(10000),b(10000)
20 for i=0 to 10000:a(i)=i:next i:rem Load an array
30 comptest off:timer=0:print "Compiler test off"
40 for i=0 to 10000:b(i)=a(i):next i
50 print "Loop executed in ";timer/50.0;" seconds"
60 comptest on:timer=0:print "Compiler test on"
70 for i=0 to 10000:b(i)=a(i):next i
80 print "Loop executed in ";timer/50.0;" seconds"
```

Try stopping the program with Control+C after the compiler tests have been switched off. The program will terminate around line 60, since this is the first time the Control+C test has been performed.

COMPTEST ALWAYS

This adds a test before each and every STOS Basic instruction. It results in slightly smoother sprite movement, and finer control over the menus. The precise effect of this command will entirely depend on the mixture of instructions in your program. If your program makes heavy use of instructions such as GOTO and FOR...NEXT, the difference will be barely noticeable. But if your routine will be performing extensive calculations while also using the sprite commands, this instruction could prove invaluable.

6: The new floating point routines

When STOS Basic was first designed, it used the latest IEEE standard for its floating point numbers. This allowed your program to use numbers between -1.797692 E+308 and +1.797693 E+307. These numbers were accurate to 16 decimal digits.

However it was quickly discovered that few users really needed this level of accuracy. The vast majority of arcade games don't use real numbers at all, and restrict themselves to integer arithmetic for the maximum speed. Furthermore, any programs which do need floating point operations usually require them to be performed extremely quickly. This is especially true of programs which generate 3D graphic effects.

After much thought, we have therefore decided to replace the existing format with the faster single precision. The new system allows a floating point number to range between 1E-14 to 1E+15,and precision is now limited to seven significant digits. This should be more than adequate for the vast majority of programmers.

The speed improvement when using the new format is extremely impressive. All floating point operations are approximately three times faster, with trigonometric functions like SIN and COS being performed at more than 30 times their earlier speed! This applies equally well to both interpreted and compiled programs.

Note that this compiler is currently **only** compatible with the new system. So unless you genuinely need to use double precision arithmetic in your programs, you should upgrade all your copies of STOS Basic to version 2.4 immediately. See the installation guide for further details of this process.

Incidentally, if you try to list any of your existing programs which use real numbers, the following text will be displayed on the screen.

```
BAD FLOAT TRAP
```

In order to allow you to run these programs from STOS v2.4 we have included a useful little utility called CONVERT.BAS which will automatically transform your programs into the correct format.

You can call this program from the compiler disc by typing:

```
run "convert.bas"
```

You will now be prompted for one of your STOS V2.3 programs. Insert the appropriate disc in drive A and select your program using the STOS file selector. This program will then be quickly converted into STOS V2.4 format, and will be copied back to the original file. It's a good idea to perform this conversion process for every one of your STOS Basic programs which use real numbers. This will avoid the risk of confusion in the future.

7: Technical details

In this section we will be discussing a range of advanced topics which will be especially relevant to those with a little programming experience.

Improved garbage collection

The problem of "garbage collection" arises in any language which allows the user to manipulate variable-sized pieces of information. The classic example of this problem in STOS Basic occurs with strings. Take the following small Basic program:

```
10 input a$:rem Input string
20 a$=a$+a$:rem Double the length of the string
30 b$=a$-" ":rem Subtract all spaces from the string
40 c$=left$(a$,3)
50 print a$
60 goto 40
```

The above program may look extremely simple, but underlying all this casual string manipulation, the STOS interpreter is performing a frantic amount of activity.

Like all variables, the characters in a string need to be stored somewhere specific in the Atari ST's memory. But what actually happens if you increase the size of a string? The system can't just tack the extra characters on at the end as this will overwrite any other variables which have been positioned immediately after it.

One solution would be to move the entire list of variables so as to create the correct number of spaces at the end of the string. In practice however, this would prove incredibly slow.

It's much easier to simply define a new string with the same name, and then insert it at the next free memory location. Of course, the characters making up the old string are now totally useless, and are taking up valuable memory space. Another source of potential waste are the intermediate results generated by operations such as "+","-",and SPACE$. As time goes by, this "garbage" will start to clutter up the ST's entire memory. Eventually, the STOS system will be forced to totally reorganize the ST's memory to recover the unused space for your program. This process is known as garbage collection.

Since it's impossible to predict when the memory will finally run out, garbage collection can occur at wildly unpredictable intervals. Furthermore, in extreme cases, the process can take up to several whole minutes to complete. This can lead to sudden and inexplicable delays in the execution of your program. The worst problems occur with programs which perform a large amount of string manipulation such as adventure games. Fortunately, the STOS compiler provides you with the perfect solution to this problem.

Enter the following program:

```
10 dim a$(5000)
20 for x=0 to 5000
30 a$(x)=space$(3)+"a"+space$(2):rem This generates a lot of garbage
40 home:print x;
50 next x
100 timer=0
110 print free:rem Force a garbage collection
120 print timer/50.0
```

If you run this prograusing STOS Basic v2.3 the garbage collection will take several minutes. Now try it on STOS Basic v2.4. You will be delighted to discover that the entire process occurs almost instantaneously. This is because, when François Lionet created this compiler, he cleverly optimised all the garbage collection routines for maximum speed. So garbage collection will never be a potential problem for one of your compiled programs.

The compiler

The STOS Basic compiler was designed to use as little memory as possible. In fact, most of the memory needed by the system is borrowed from the sprite background screen. That's why the mouse disappears while the compiler is executing.

How the compiler works

The compiler first reserves some memory in the current background screen. It then looks into the COMPILER folder and opens the main Basic library (BASIC204.LIB). The addresses of all the appropriate library routines are now loaded into the ST's memory. The next action, is to check for the existence of an extension file ending in .EC. These contain all the information required to compile the BASIC commands added by an extension. Whenever an extension is discovered, a full catalogue of the additional routines is then added to the current list. The execution of the compiler is split into three separate phases which are known as passes.

PASS 0 The first pass checks the STOS Basic program for possible syntax errors, and makes an initial attempt at converting the program into machine code. While it does this, it produces a full list of all the library routines which will be called by the program. Note that no actual code is actually generated by this pass as the intention is merely to estimate the final size of the compiled program. The compiler can now safely reserve the precise amount of space which will be needed, without wasting valuable memory.

PASS 1 Analyses the Basic program using exactly the same method as pass 0. It then converts the entire STOS Basic program into machine code, and copies this data to either memory or the disc. At the same time it also creates a number of tables including the relocation table.

The compiler now incorporates any library routines which are accessed from the compiled program. It is important to note that only the routines which are actually used by the program will be included in the final code. This reduces the size of the compiled program to the absolute minimum. The following steps are then performed by the compiler in quick succession:

1 If an extension command is used in the program, the extension libraries are searched and the appropriate routines are written into the compiled program.
2 The relocation table is copied into the program. This allows the compiled program to be executed anywhere in the ST's memory.
3 The table used to hold the address of the program lines is then added.
4 Any string constants which are used are tagged onto the end of the program.
5 If the program is to be run from Gem, the compiler copies over the various library routines needed by the sprites, windows, menus, music and floating point arithmetic. These add approximately 40k to the length of the program

PASS 2 This pass simply explores the relocation table created in pass 1, and sets the required addresses in the compiled program. The compiler now closes all the open disc files, and transfers the program to the current program segment if required.

Note that the eventual size of a compiled program depends entirely on the precise mix of STOS instructions which are used. There's no real relationship between the complexity of the program and the size of the code. In practice, some of the simplest Basic instructions proved to be the hardest to actually write. A good example of this is the STOS file selector, which involves over 4k of machine code.

You can see below the machine code produced by the compiler from a simple plot command. The Basic listing:

```
plot 320,100,1
```

The compiled program:

```
move.l #1,-(a6)
move.l #100,-(a6)
move.l #320,-(a6)
jsr plot
```

The subroutine 'plot' is a library routine which will be merged into the compiled program.

STOS-run

STOS-run programs have the standard Basic header and fake line at 65535 with a single instruction which calls the compiled program. The memory banks are handled by the editor in exactly the same way as for a normal interpreted program. This means you can BGRAB or SAVE them in the usual way. Incidentally, it's also possible to execute a compiled program as a STOS accessory. In order to do this, load them into memory and resave them with the extension ".ACB".

Gem-run

GEM-run programs have the same header format as TOS files. There is, however, **no relocation table** for TOS, and the relocation address points to an empty table. Instead of this, the beginning of the program is written in PC relative code. This contains a small routine which relocates the main program using the STOS relocation table.

The first thing a GEM-run program does is to initialise the memory banks which are normally chained to the compiled program. It starts by finding the address of the top of memory using location $42E. It then subtracts 64k for the screens and moves the memory banks (if present) to the end of memory. This process provides the compiled program with plenty of free space to work with, situated between the end of the program and the beginning of the memory banks.

The program then sets up the standard STOS environment. It first initialises all the TRAP routines (sprites, windows, music). Then it activates the normal STOS interrupts and kills the interrupts used by GEM.

Finally it erases the screen, activates the mouse pointer and starts executing the compiled program.

8: Utility programs

This package includes a number of small accessory programs for your use.

The ramdisc accessory

The STOSRAM program allows you to create a ramdisc of any size for up to the maximum available ram. It is especially useful for 1040 users who can copy the COMPILER folder into memory, speeding up the compilation process significantly. See the section on installation for more details.
The action of the accessory is to add a separate STOSRAM.PRG program to your current AUTO folder. This will be executed every time STOS Basic is subsequently run, and will be automatically loaded with the contents of any folder on the current disc. Here is a list of the possible options.

<A> or <B>	Sets the drive on which the ramdisc program will be installed.
<S>	Chooses the size of the ramdisc. The default is 150k, which is just right for the contents of the COMPILER folder. When this option is selected you will be requested to input the ramdisc's size. This number should be entered in units of a kilobyte.
<C>	Selects the path name for the folder which is to be loaded into theramdisc on start-up. If this string is empty, or the folder you have requested cannot be found, then the ramdisc will be left vacant. Note that only he individual files in the directory can be copied, **not** entire folders.
<G>	Creates a new ramdisc using the options you have previously set. TheSTOSRAM.PRG program is now copied into the current AUTO folder onthe disc. If an AUTO folder doesn't currently exist, then one will be created. An important point to remember is that the ramdisc won't be removed from memory by resetting the computer– it must be completelyturned off. If you reset and boot up STOS another ramdisc will becreated.

The disc formatter accessory

The FORMAT accessory enables you to format a disc directly from STOS Basic. Discs can be formatted using the following options:

<A> or <B>	Selects current drive.
<1> or <2>	This toggles between 360k single sided format (1) and 720k doublesided format (2).
<G>	Formats the disc in the current drive.